Formal Verification of the *rank* Algorithm for Succinct Data Structures ⋆

Akira Tanaka[1]   Reynald Affeldt[1]   Jacques Garrigue[2]

[1] National Institute of Advanced Industrial Science and Technology (AIST)
[2] Nagoya University

Abstract. Succinct data structures are designed to use a minimal amount of computer memory in a time-efficient way. Their correct implementation is essential to big data analysis. Yet, succinct data structures are difficult to verify because they rely on bit-level manipulations better achieved with low-level languages. In this paper, we report on the formal verification of the standard Jacobson *rank* algorithm using the Coq proof-assistant and extract an OCaml implementation from it. This requires overcoming the mismatch between Coq being a purely functional programming language and succinct data structures being inherently imperative. To enjoy the best of both worlds, we propose to use code extraction from Coq to OCaml but with an original (tested but unverified) implementation of bitstrings. We can then use Coq to formalize correctness, including important claims about storage requirements, and still obtain efficient native code. To the best of our knowledge, this is the first application of formal verification to succinct data structures.

1 Towards Formal Verification for Succinct Data Structures

Succinct data structures are data structures designed to use an amount of computer memory close to the information-theoretic lower bound in a time-efficient way (see [18] for an introduction). They are used in particular to process big data. Concretely, succinct data structures make it possible to provide data analysis with a significantly reduced amount of memory (for example, one order of magnitude less memory for string search facilities in [2]). Thanks to an important amount of research, succinct data structures are now equipped with algorithms that are often as efficient as their classical counterparts. In this paper, we are concerned with the most basic one: the *rank* algorithm, which counts the number of 1 (or 0) in the prefixes of a bitstring (for example, *rank* is one of the few basic blocks in the implementation of [2]—see appendix A of the technical report). The salient property of the *rank* algorithm is that it requires $o(n)$ storage for constant-time execution where n is the length of the bitstring (see Sect. 2 for background information).

Our long-term goal is to provide formal verification of algorithms for succinct data structures. In particular, we aim at the construction of a realistic library of verified algorithms. Such a library could significantly improve the confidence in software implementation of big data analysis. However, software implementations of algorithms

⋆ This is preprint with appendix of a paper to be presented at ICFEM 2016: `http://icfem2016.xyz/`

for succinct data structures are difficult to verify. Indeed, since these data structures are designed at the bit-level and since performance is a must-have, they are usually written in low-level languages (e.g., C++ for SDSL [16]). The direct verification of C-like languages is now possible [14] but it requires a substantial infrastructure (concretely, an instrumented formal semantics of the target language) whose development is orthogonal to the problem of verifying succinct data structures.

In this paper, we show how to develop a verified implementation of an algorithm for succinct data structures using the Coq proof-assistant [5]. Coq provides us with the ability to reason about the correctness of the algorithm: its functional correctness but also the important properties about storage requirements. We can also derive an efficient implementation thanks to the extraction facility from Coq to the OCaml language and the imperative features of the latter. The main issue when dealing with algorithms for succinct data structures in Coq is that, since Coq is a purely functional language, arrays are better represented as lists to perform formal verification. However, lists do not enjoy constant-time random-access, making it difficult to use the extraction facility of Coq to generate efficient OCaml algorithms. As a solution, we provide an OCaml library for bitstrings with constant-time random-access that matches the interface of Coq lists so that we can use real bitstrings in the extracted code. This approach augments the trusted base but in the form of a localized, reusable library of OCaml code whose formal verification can anyway be carried out at a later stage. We think that this is a reasonable price to pay compared to the benefits of carrying out formal verification in Coq.

Paper Overview In this paper, we demonstrate our approach by building a verified implementation of the *rank* function using the Coq proof-assistant. More precisely, we provide formal verification for the *rank* function (formal proof of functional correctness in Sect. 5.2 and formal proof for storage requirements in Sect. 5.3) and extraction to executable OCaml code (by providing in particular a new library for bitstrings with constant-time random-access in Sections 4.2 and 4.3). We will be able to check that the time-complexity of the extracted code is as expected (i.e., execution is constant-time, see Sect. 6.2). In the process, we discuss thoroughly the choices we made, in particular, the modular approach we took when formalizing the *rank* function in the Coq proof-assistant (generic version in Sect. 3.1 and its instantiation in Sect. 5.1).

2 A Formal Account of the *rank* Algorithm

We explain what the *rank* algorithm is supposed to achieve (its *functional correctness*, Sect. 2.1) and how Jacobson's *rank* actually achieves it (in particular, its storage requirements, Sect. 2.2). These points are addressed formally using Coq resp. in Sections 5.2 and 5.3.

2.1 Specification of the Functional Correctness of the *rank* Algorithm

Given a bitstring s and an index i in s, $\mathit{rank}_s(i)$ counts the number of 1's up to i (excluded). For example, in Fig. 1 (the first and second-level directories will be explained in Sect. 2.2), s contains 26 1's, $\mathit{rank}_s(4) = 2$, $\mathit{rank}_s(36) = 17$, and $\mathit{rank}_s(58) = 26$.

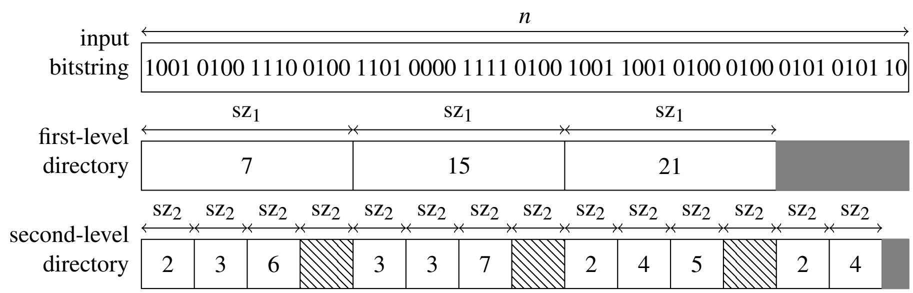

Fig. 1. Illustration for the *rank* algorithm ($\mathrm{sz}_2 = 4$, $\mathrm{sz}_1 = 4 \times \mathrm{sz}_2$, $n = 58$). Example extended from [13].

The mathematically-inclined reader would formally specify the *rank* algorithm as $rank_s(i) = |\{k \in [0,\ldots,i) \,|\, s[k] = b\}|$ where b is the query bit ($b = 1$ in the example above). Using the Coq proof-assistant, such a specification can be formalized directly. For bits, one can use the Coq type for booleans `bool`. An input bitstring can be formalized as a list of booleans (type `seq bool` in Coq). An index i is a natural number (type `nat` in Coq). A functional programmer would formally specify the *rank* algorithm as list surgery and filtering. For example:

```
Definition rank b i s := count_mem b (take i s).
```

We regard this Coq function as the specification of the functional correctness of the *rank* algorithm. Note that it does not provide an efficient implementation: it can be executed (both in Coq and as an extracted OCaml program) but computation would (hopefully) be linear-time. In this paper, we provide Coq functions that are more realistic in the sense that they can be extracted to executable OCaml code.

2.2 Jacobson's *rank* Algorithm and Its Space Complexity

Jacobson's *rank* algorithm [11] is a constant-time implementation of *rank*. It uses auxiliary data structures, in particular two arrays called the first and second-level directories that essentially contain pre-computed values of *rank* for substrings of the input bitstring s of size n (see Fig. 1). More precisely, each directory contains fixed-size integers, whose bit-size is large enough to represent the intended values, so that the bit-size for each directory depends on n.

Let sz_2 be the size of the substrings used for the second-level directory. Hereafter, we refer to these substrings as the "small blocks". The size of the substrings used for the first-level directory is $\mathrm{sz}_1 = k \times \mathrm{sz}_2$ for some k. We refer to these substrings as the "big blocks". The first-level directory is precisely an array of n/sz_1 integers such that the ith integer is $rank_s((i+1) \times \mathrm{sz}_1)$. The second-level directory is also an array of integers. It has n/sz_2 entries and is such that the ith entry is the number of bits among the $(i\%k+1) \times \mathrm{sz}_2$ bits starting from the $((i/k) \times \mathrm{sz}_1)$th bit ($/$ is integer division and $\%$ is the remainder operation). One can observe that when $i\%k = k-1$, the ith entry of the second-level directory (the hatched rectangles in Fig. 1) can be computed from the first-level directory and therefore does not need to be remembered.

Given an index i, Jacobson's *rank* algorithm decomposes i such that $rank_s(i)$ can be computed by adding the results of (1) one lookup into the first-level directory, (2) one lookup into the second-level directory, and (3) direct computation of *rank* for a substring shorter than sz_2. For example, in Fig. 1, $rank_s(36) = rank_s(2\times 16 + 1\times 4 + 0)$ is computed as $15+2$ and $rank_s(58) = rank_s(3\times 16+2\times 4+2)$, as $21+4+1$. Since the computation of *rank* for a substring shorter than sz_2 in (3) can also be tabulated or computed with a single instruction on some platforms, *rank*'s computation is constant-time.

It can be shown (and we will do it formally in Sect. 5.3) that the directories require only $\frac{n}{\log_2 n} + \frac{2n\log_2\log_2 n}{\log_2 n} \in o(n)$ bits with integers of the appropriate size (not necessarily the word size of the underlying architecture).

3 Our Approach: Extraction From a Generic *rank* Function

In a nutshell, our approach consists in (1) providing a generic implementation of the *rank* algorithm to keep formal proofs as high-level as possible and (2) extracting OCaml code from a concrete instantiation of the *rank* algorithm. As explained in Sect. 1, this approach makes it difficult to obtain efficient OCaml code because of the conflicting requirements between the data structures at the formal proof level and at the implementation level. We make this idea clearer in Sect. 3.2 where we also justify our approach. Before that, we explain the (generic) *rank* algorithm that we will verify and extract (instantiation to be found in Sect. 5.1).

3.1 A Generic Rank Algorithm Formalized in Coq

The generic version essentially consists of two functions: one that constructs the directories and one that performs the lookup.

To simplify the presentation, we first explain a function that counts bits in a naive way[3]. `bcount b i l s` counts the number of bits `b` (0 or 1) inside the slice `[i,...,i + l)` of the bitstring `s` (essentially a list of booleans—see Sect. 4.1):

```
Definition bcount b i l s := count_mem b (take l (drop i s)).
```

In the code below, we use notations from the Mathematical Components [7] library: `.+1` is the successor function, `%/` and `%%` are the integer division and modulo operators, and `if x is xp.+1 then e1 else e2` means: if `x` is greater than 0 then return `e1` with `xp` bound to $\mathtt{x} - 1$, else return `e2`.

Construction of the Directories The function `buildDir` computes both directories in one pass (it returns a pair). It has been written with extraction in mind. In particular, it uses tail calls, and indexing instead of list pattern-matching.

`j` is a counter for small blocks (we start counting from `nn`, the total number of small blocks, i.e., n/sz_2). `i` is a counter to count small blocks in one big block. `n1` contains the

[3] The function `bcount` is not intended to be extracted as it is but replaced by a more efficient function. It could be tabulated as explained in Sect. 2.2, but in this paper, it will be replaced by a single gcc built-in operation (see Sect. 4.2).

number of bits counted so far for the current big block. `n2` contains the number of bits counted so far for the current small block. `D1` (resp. `D2`) are abstract data types meant for the first-level (resp. second-level) directory (so that `emptyD1`, `pushD1`, etc. are meant to be instantiated with concrete functions later).

The function `buildDir` iterates over the number of small blocks. At each iteration, the number of bits in the current small block is stored in `m` (line 2) (`b` is the query bit, `sz2` is the size of small blocks, `inbits` is the input bitstring). For each small block, `n2` is stored in the second-level directory (line 4). After a big block has been scanned, the number of bits counted so far for the current big block `n1 + n2` is stored in the first-level directory (line 8). The number of small blocks in one big block (`kp` plus 1) is used to control the iteration inside a big block (line 10).

Observe that the directories built by `buildDir` are slightly different from the data structures explained in Sect. 2.2: they start with a 0 (stored at line 8 for the first-level directory and stored at line 9 for each group of small blocks) which is of course not necessary but this simplifies the lookup function.

```
1  Fixpoint buildDir j i n1 n2 D1 D2 :=
2    let m := bcount b ((nn - j) * sz2) sz2 inbits in
3    if i is ip.+1 then
4      let D2' := pushD2 D2 n2 in
5      if j is jp.+1 then buildDir jp ip n1 (n2 + m) D1 D2'
6      else (D1, D2')
7    else
8      let D1' := pushD1 D1 (n1 + n2) in
9      let D2' := pushD2 D2 0 in
10     if j is jp.+1 then buildDir jp kp (n1 + n2) m D1' D2'
11     else (D1', D2').
12 Definition rank_init_gen := buildDir nn 0 0 0 emptyD1 emptyD2.
```

Lookup The function `rank_lookup_gen` is a generic implementation of the lookup function. It computes the rank for index `i`:

```
Definition rank_lookup_gen i :=
  let j2 := i %/ sz2 in (* index in the second-level directory *)
  let j3 := i %% sz2 in (* index in a small block *)
  let j1 := j2 %/ k in (* index in the first-level directory *)
  lookupD1 j1 D1 + lookupD2 j2 D2 + bcount b (j2 * sz2) j3 inbits.
```

`j1` (resp. `j2`) is the index of the block in the first-level directory (resp. second-level directory). They are computed using the size of small blocks `sz2` and the ratio between the size of big and small blocks `k` (or in other words, `sz1 = k * sz2`). `lookupD1` (resp. `lookupD2`) is meant to perform array lookup; it will be instantiated later.

3.2 Our Approach w.r.t. Extraction

In the code above, lookup in the directories is meant to be performed by the functions `lookupD1` and `lookupD2`. Constant-time execution for these functions is required for Jacobson's *rank* function to be efficient. If we implement these functions with `nth`-like access to standard `lists` (which is linear-time), Coq will not generate OCaml functions with the desired time complexity. At first, one may think of looking for an ingenious implementation scheme that may cause Coq to generate efficient OCaml code. This approach seems to us too optimistic as a first step towards the goal of providing a verified library of functions for succinct data structures for the following two reasons:

- Coming up with new implementation schemes is likely to make more difficult the task of proving formally the functional correctness and the storage requirements of algorithms.
- The code extraction facility of Coq is not optimized in any way (by design, because it is part of the trusted base). In practice, it tends to generate inefficient code for convoluted formalizations. As a matter of fact, previous work shows that Coq requires significant engineering to handle imperative features and native data structures (e.g., [3]).

Instead, our approach consists in (1) making the best we can out of list-like data structures in Coq and (2) providing an efficient OCaml implementation of the list interface that we will substitute to Coq-generated functions.

4 An OCaml Bitstring Library for Coq Lists of Booleans

Direct extraction of Coq lists and list functions suffers two major problems w.r.t. succinct data structures: (1) memory usage is very inefficient (assuming 64-bit machine words, it would take 192 bits to represent one boolean), (2) random-access will be linear-time instead of the required constant-time complexity. We now explain an OCaml implementation for the interface of Coq lists that solves above problems.

4.1 Bitstrings Formalized in Coq

We define bitstrings as an inductive type which wraps Coq lists:

```
Inductive bits : Type := bseq of seq bool.
```

The type `bits` is isomorphic to the type of lists of booleans. In consequence, many functions for `bits` are easily derivable from Coq standard functions `size`, `nth`, `++` (concatenation), etc. In particular, we equip our formalization with a *coercion* that transparently turns the type `bits` into the type `seq bool`. Concretely, this coercion is the function `Definition seq_of_bits s := match s with bseq l => l end.` that is automatically inserted by Coq to make types match. For example, `size s` below should actually read as `size (seq_of_bits s)`.

```
Definition bnil := bseq nil.
Definition bsize (s : bits) := size s.
Definition bnth (s : bits) i := nth false s i.
Definition bappend (s1 s2 : bits) := bseq (s1 ++ s2).
```

However, code extracted from above functions does not achieve the desired complexity. For example, the code extracted from `bsize`, `bnth`, and `bcount` (Sect. 3.1) would be linear-time because these functions scan the lists obtained from `bits`[4]. Regarding memory usage, the list constructor `cons` would allocate one memory block per argument (see Fig. 2, on the left, for an illustration). In addition, OCaml needs one more word for each block to manage memory. Assuming the machine word is 64 bits, `cons` would therefore need 192 bits to represent a Coq `bool`, that was supposed to represent a single bit...

In the next section (Sect. 4.2), we provide OCaml definitions to replace the Coq type `bits`, its constant `bnil` and the functions `bsize`, `bnth`, `bappend`, etc. How the OCaml definitions are substituted for the Coq definitions is explained in Sect. 6.1.

4.2 Bitstrings Implemented in OCaml

The main idea to achieve linear-time construction and constant-time random-access in OCaml is to implement bitstrings using a datatype that allows for random-access of bits. For this purpose, we use the type `bytes` introduced in OCaml 4.02.0 [5]. The resulting OCaml type is as follows[6]:

```
type bits_buffer = { mutable used : int; data : bytes; }
type bits = Bdummy0 | Bdummy1 | Bref of int * bits_buffer (* len, buf *)
```

Bitstrings are stored in a `bits_buffer` as a value of type `bytes` together with the number of bits `used` so far. (The first bit is the least significant bit of the first byte in the `bytes`.) Let us first explain the constructor for arbitrary-length bitstrings (`Bref`) and then explain how short bitstrings are implemented as unboxed integers (this will explain `Bdummy0` and `Bdummy1`).

`bits` represented with `Bref` The data structure `Bref`(len, buf) (depicted on the right of Fig. 2) represents the prefix of size *len* of the bitstring *buf*. Let us call *used* the value of the field `used` of the corresponding `bits_buffer` data structure.

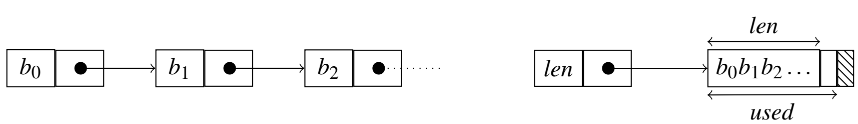

Fig. 2. A Coq `bits` on the left and the corresponding OCaml `bits` on the right

The dynamics of `Bref` is as follows. Initially, a `Bref` has 0 as *len* and references a `bits_buffer` with *used* as 0, which means that the bitstring is empty. When a bit is appended to the `Bref`, the `bits_buffer` is destructively updated and a new `Bref` is allocated. The bit is assigned to the *used*th bit in `data` and *used* is incremented. A new `Bref` is allocated with incremented *len* and reference the `bits_buffer`. (When the

[4] Let `s` be a bitstring of length n. `bsize s` is $O(n)$, `bnth i s` is $O(i)$, `bcount b i l s` is $O(i+l)$. `bcount` requires an additional $O(i)$ because of the `drop` function (see Sect. 3.1).

[5] Currently, `bytes` is the same as `string`; OCaml plans to change `string` to immutable.

[6] The OCaml definitions below belong to the module `Pbits`; the prefix `Pbits.` is omitted when no confusion is possible.

bits_buffer is full (i.e., $8 \times |\texttt{data}| = used$), data is copied into a new bytes with a doubled length before the bit is appended.) Array construction always append a bit to Bref which *len* is equal to *used*.

The constructor Bref can represent any bitstring but it requires memory allocation for each value, even to represent an empty bitstring, a single boolean, etc. We can improve efficiency by avoiding memory allocation for bitstring that fit in machine words. Note that there is no soundness problem in losing sharing of bitstrings, because bitstrings bits are immutable in Coq.

bits represented with unboxed integers In summary, we use the unboxed integers of OCaml to represent short bitstrings. In OCaml, values are represented by w-bit integers, w being the number of bits in a machine word (32 or 64). These integers represent either (1) a $(w-1)$-bit unboxed integer or (2) a pointer to a block allocated in the heap. OCaml datatypes use unboxed integers for constant constructors, and pointers to blocks otherwise. Therefore, we can represent short bitstrings by unboxed integers. More precisely, we represent bitstrings of length $u \leq w-2$ as a $(w-1)$-bit integer using the following format: $\overbrace{0\ldots0}^{w-u-2}1b_{u-1}\ldots b_1b_01$ (the position of the topmost 1 represents the length of the bitstring and the trailing 1 is a tag bit to distinguish unboxed integers from pointers). To treat the latter integers as bits we use Obj.magic. For example, bnil $(0\ldots011)$ is defined as follows.

```
let bits_from_int bn = ((Obj.magic (bn : int)) : bits)
let bnil = bits_from_int 1 (* the tag bit is invisible in OCaml *)
```

The reason for adding the constructors Bdummy0 and Bdummy1 to the datatype bits is technical. Without them, OCaml optimizes pattern-matching (discrimination of values with match) if a datatype has no constant constructor (assuming that the value must be a pointer), or if it has only one constant constructor (assuming that any non-zero value must be a pointer). Adding two constant constructors disables these optimizations, and allows us to safely use pattern-matching to discriminate unboxed integers from Bref blocks.

OCaml Functions for Bitstrings Using the OCaml bits datatype, we have implemented OCaml functions that match the Coq functions of Sect. 4.1 but with better complexities, as summarized in Table 1. For this purpose, we make use of OCaml imperative features such as destructive update and random access in bytes. Details about the OCaml implementation can be found in appendix A.

Table 1. Time complexity of OCaml functions w.r.t. their Coq counterparts (n and n' are the lengths of s and s')

Function	Complexity in Coq	Complexity in OCaml
bsize s	$O(n)$	$O(1)$
bnth s i	$O(i)$	$O(1)$
bappend s s'	$O(n)$	$O(n')$ (amortized, for array construction)
bcount b i l s	$O(i+l)$	$O(l)$

4.3 From Natural Numbers to Fixed-size Integers

At the abstract level, the *rank* algorithm stores natural numbers in directories but a concrete implementation manipulates fixed-size integers instead. For this reason, we extend our Coq formalization and OCaml implementation of bitstrings with functions to manipulate fixed-size integers:

- `bword u n` builds a short bitstring from the lower `u` $\leq w-2$ bits of a natural number `n` in constant-time. In OCaml, a natural number is formatted as $b_{w-2}\dots b_1 b_0 1$, where w is the number of bits in a machine word. In order to construct short bitstrings as unboxed integers following the format explained in Sect. 4.2, we use simple bit operations: clear the higher bits, $b_{w-2}\dots b_{u+1}$, and set the topmost bit, b_u.
- `getword i u s` looks for the `u` $\leq w-2$ bits (ordered with least significant bit first) starting from index `i` in `s`, regarding them as a natural number. In OCaml, this function is implemented by accessing data at the level of bytes (not bits) to reduce the overhead (number of bit operations and number of loops).

Using these functions, it becomes possible to provide a concrete instantiation of directories. For example, let us consider the first-level directory, that stores fixed-size integers of size `w1`. Its implementation is summarized in Table 2. Let `D1Arr` be the type of the first-level directory. An empty first-level directory is implemented by an empty array `emptyD1` that is just an empty bitstring `bnil`. The result of appending an unboxed integer `n` (seen as a `w1`-bit bitstring) to the first-level directory `s` is implemented by the array `pushD1 w1 s n`. `lookupD1 w1 i s` is the `i`th pushed in the first-level directory `s`.

Table 2. Interface and implementation of the first-level directory using generic array functions

Interface	Implementation
`D1Arr`	`bits`
`emptyD1 : D1Arr`	`bnil`
`pushD1 w1 s n : D1Arr`	`bappend s (bword w1 n)`
`lookupD1 w1 i s : nat`	`getword (i * w1) w1 s`

5 Formal Verification of an Instance of the Generic *rank* Algorithm

We instantiate the generic *rank* function of Sect. 3.1 to obtain a concrete implementation of Jacobson's *rank* algorithm. Then, we prove that this implementation indeed computes *rank* (as specified in Sect. 2.1) and fulfills storage requirements (as seen at the end of Sect. 2.2).

5.1 Instantiation of the *rank* Algorithm

We instantiate the functions from Sect. 3.1 (`rank_lookup_gen` and `rank_init_gen`) with the array of bits from Sect. 4.3. The parameters of this instantiation (number and size of blocks in the directories, etc.) are important because they need to be properly set to achieve the storage requirements specified in Sect. 2.2. For the sake of clarity, we isolate these parameters by means of two datatypes. `Record Param` carries the parameters of Jacobson's algorithm. `Record Aux` essentially carries the results of the execution of the initialization phase:

```
1  Record Param : Set := mkParam
2  { kp_of : nat ;
3    sz2p_of : nat ;
4    nn_of : nat ;
5    w1_of : nat ;
6    w2_of : nat }.
7  Record Aux : Set := mkAux
8  { query_bit: bool;
9    input_bits: bits;
10   parameter: Param;
11   directories: D1Arr * D2Arr }.
```

Jacobson's algorithm is parameterized by the number of small blocks (minus 1) in a big block (or $sz_1/sz_2 - 1$) (field `kp_of`, line 2), the number of bits (minus 1) in a small block (or $sz_2 - 1$) (line 3), the number of small blocks (line 4), and the bit-size of fixed-size integers for each directory (lines 5–6). The instantiation of `rank_init_gen` returns the query bit (line 8), the input bitstring (line 9), the parameters of Jacobson's algorithm (line 10), the first and second-level directories themselves (line 11).

The instantiation of `rank_init_gen` is a matter of passing the appropriate `parameters` and the functions `D1Arr`, `D2Arr`, etc. that we explained in Sect. 4.3:

```
Definition rank_init b s : Aux :=
  let param := rank_param (bsize s) in
  let w1 := w1_of param in let w2 := w2_of param in
  mkAux b s param
    (rank_init_gen b s param
      D1Arr emptyD1 (pushD1 w1) D2Arr emptyD2 (pushD2 w2)).
```

Similarly, `rank_lookup_gen` is instantiated with the `parameters` resulting from the execution of `rank_init` together with the functions `D1Arr`, `D2Arr`, etc. from Sect. 4.3:

```
Definition rank_lookup aux i :=
  let b := query_bit aux in
  let param := parameter aux in
  let w1 := w1_of param in let w2 := w2_of param in
  rank_lookup_gen b (input_bits aux) param
    D1Arr (lookupD1 w1) D2Arr (lookupD2 w2)
    (directories aux) i.
```

5.2 Functional Correctness of Jacobson's Algorithm in Coq

The functional correctness of Jacobson's algorithm is stated using the generic *rank* function (`rank_lookup_gen`, Sect. 3.1) with its formal specification (`rank`, Sect. 2.1). As a matter of fact, we do not need to assume any concrete instantiation of the directories to establish functional correctness, the generic properties of arrays are sufficient.

```
Lemma rank_lookup_gen_ok_to_spec : forall i dirpair,
  i <= size inbits ->
  dirpair = rank_init_gen b inbits param
```

```
    D1Arr emptyD1 pushD1 D2Arr emptyD2 pushD2 ->
  rank_lookup_gen b input_b param
    D1Arr lookupD1 D2Arr lookupD2 dirpair i = rank b i inbits.
```

The many parameters `D1Arr`, `D2Arr`, etc. come from the array interface that we implemented using the `Section` mechanism of Coq.

5.3 Space Complexity of Auxiliary Data Structures

The required storage depends on the parameters of Jacobson's algorithm explained in Sect. 5.1. They should be chosen appropriately to achieve $o(n)$ space complexity. We use the following parameters. They are taken from [4, Sect 2.2.1]. We add 1 to sz_2 and k to make them strictly positive for all $n \geq 0$.

$$\begin{aligned} k &= \lceil \log_2(n+1) \rceil + 1 & w_1 &= \lceil \log_2(\lfloor n/\mathrm{sz}_2 \rfloor \times \mathrm{sz}_2 + 1) \rceil \\ \mathrm{sz}_2 &= \lceil \log_2(n+1) \rceil + 1 & w_2 &= \lceil \log_2((k-1) \times \mathrm{sz}_2 + 1) \rceil \\ \mathrm{sz}_1 &= k \times \mathrm{sz}_2 = (\lceil \log_2(n+1) \rceil + 1)^2 \end{aligned}$$

The formalization in Coq of above parameters is direct. Below, `bitlen` n[7] is Coq code for $\lceil \log_2(n+1) \rceil$:

```
Definition rank_default_param n :=
  let kp := bitlen n in (* k-1 *)
  let sz2p := bitlen n in (* sz2-1 *)
  let sz2 := sz2p.+1 in
  let nn := n %/ sz2 in
  let w1 := bitlen (n %/ sz2 * sz2) in
  let w2 := bitlen (kp * sz2) in
  mkParam kp sz2p nn w1 w2.
```

Using these parameters, we showed that the asymptotic storage requirement for the auxiliary data structures is indeed $o(n)$, more precisely $\frac{n}{\log_2 n} + \frac{2n \log_2 \log_2 n}{\log_2 n}$, similarly to [4, Theorem 2.1].

For the sake of illustration, let us show how we prove in Coq that the contribution of the first-level directory to space complexity is $\frac{n}{\log_2 n}$. First, we fix *rank*'s parameters using the following declaration:

```
Definition rank_param n := rank_param_w_neq0 (rank_default_param n).
```

`rank_default_param` has been explained just above. `rank_param_w_neq0` is just a technicality to take care of the uninteresting case where the length of input bitstring is zero[8]. The contribution of the first-level directory to space complexity is the length of the bitstring that represents it, i.e., `size (directories (rank_init b s)).1` (.1 stands for the first projection of a pair). In Coq, we proved the following lemma about this length:

[7] This function is implemented in C using gcc's `__builtin_clzl` [6], which counts the number of leading zeros in a `long` value. gcc generates LZCNT instructions (since Intel AVX2 [8]).

[8] In this case, `w1` and `w2` become 0 and our word array cannot distinguish an empty array and non-empty array.

```
Lemma rank_spaceD1 b s :
  size (directories (rank_init b s)).1 =
  let n := size s in let m := bitlen n in
  ((n %/ m.+1) %/ m.+1).+1 * (bitlen (n %/ m.+1 * m.+1)).-1.+1.
```

(`.-1` is notation for the predecessor function.)

For the sake of readability, we write this Coq expression using mathematical notations (in the case where $n \geq 3$):

$$\left(\frac{\frac{n}{\lceil \log_2(n+1) \rceil + 1}}{m+1} + 1\right) p \qquad \begin{array}{l} \text{with :} \\ m = \lceil \log_2(n+1) \rceil \\ p = \lceil \log_2(\frac{n}{m+1} \cdot (m+1) + 1) \rceil \\ \text{where } \div \text{ is the Euclidean division} \end{array}$$

When n is large, we observe that $m \sim p$, thus the whole expression is asymptotically equal to $\frac{n}{\log_2 n}$, as desired. See [19] for the $\frac{2n \log_2 \log_2 n}{\log_2 n}$ contribution of the second-level directory to space complexity.

6 Final Extraction and Benchmark

We extract the *rank* function from Sect. 5.1 using the OCaml library for bitstrings from Sect. 4.2 and benchmark the result to check that its execution is constant-time.

6.1 Extraction of the Verified *rank* Function

Concretely, extraction from Coq is the matter of the command `Extraction` (see file `Extract.v` [19]).

Extraction of Coq Lists To replace inductive types and functions with custom OCaml code, we provide the following hints:

```
1 Extract Inductive bits =>
2   "Pbits.bits" [ "Pbits.bseq" ] "Pbits.bmatch".
3 Extract Inlined Constant bnil => "Pbits.bnil".
4 Extract Inlined Constant bsize => "Pbits.bsize".
5 Extract Inlined Constant bnth => "Pbits.bnth".
6 Extract Inlined Constant bappend => "Pbits.bappend".
```

At line 1, we replace the Coq inductive type `bits` with the OCaml type `Pbits.bits` defined in OCaml. `Pbits.bseq` and `Pbits.bmatch` are specified to replace the constructor and pattern-matching expression which converts list of booleans to `Pbits.bits` and vice versa. `Pbits.bseq` and `Pbits.bmatch` are defined but our application doesn't use them to avoid memory-inefficient list of booleans.

From line 3, the constant and functions `bnil`, `bsize`, `bnth`, etc. from Sect. 4.1 are replaced by `Pbits.bnil`, `Pbits.bsize`, `Pbits.bnth` etc. to be explained in Sect. 4.2.

Extraction of the *rank* Algorithm Because we used abstractions in Coq, we must be careful about inlining at extraction-time to obtain OCaml code as efficient as possible. In particular, we need to ensure that the function parameters we have introduced for modularity using Coq's `Section`s are inlined. Concretely, we inline most function calls using the following Coq command: `Extraction Inline emptyD1 pushD1 lookupD1 ...` . As a result, `rank_lookup` looks like an hand-written program, prefix notations aside (see appendix B for the extracted `rank_lookup` and `rank_init` functions). As for the function `buildDir` in `rank_init`, we obtain a tail-recursive OCaml function, like the one we wrote in Coq, so that it should use constant-size stack independently of the input bitstring.

Since we obtain almost hand-written code, we can expect `ocamlopt` to provide us with all the usual optimizations. There are however specific issues due to Coq idiosyncrasies. For example, the pervasive usage of the successor function `.+1` for natural numbers is extracted to a call to the OCaml function `Pervasives.succ` that `ocamlopt` luckily turns into an integer increment. (One can check which inlining `ocamlopt` has performed by using `ocamlopt -dclambda`.) In contrast, anonymous function calls produced by extraction may be responsible for inefficiencies. For example, the mapping from Coq `nat` to OCaml `int` is defined as follows (file `ExtrOcamlNatInt.v` from the Coq standard library) :

```
Extract Inductive nat => int [ "0" "Pervasives.succ" ]
 "(fun fO fS n -> if n=0 then fO () else fS (n-1))".
```

It is responsible for calls of the form `(fun fO fS n -> ...) (fun _ -> E1) (fun jp -> E2)` (see `rank_init` in appendix B) that `ocamlopt` unfortunately cannot β-reduce.

6.2 Benchmarking of the Verified *rank* Function

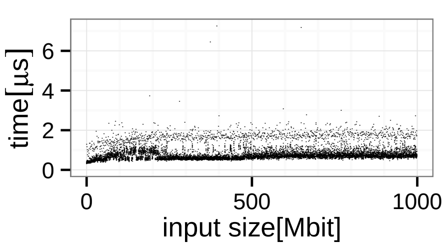

Fig. 3. Performance of rank lookup

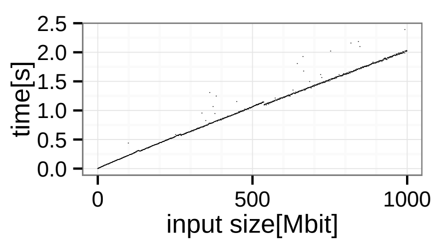

Fig. 4. Performance of rank initialization

Fig. 3 shows the performance of a single lookup invocation for the *rank* function by measuring the time taken by `rank_lookup aux i` for inputs up to 1000Mbit (recall that the input string `s` is part of `aux`). We make measurements for 1000 values of the input size n. For each n, we make 10 measures for `i` between 0 and n. The measurement order is randomized (n and `i` are picked randomly).

Execution seems constant-time (0.83µs on average) w.r.t. the input size. One can observe that execution seems a bit faster for small inputs. We believe that this is the effect of memory cache. One can also observe that the result is noisy. We believe that this is because of memory cache with access patterns and some instructions, such as IDIV (integer division), that use a variable number of clock cycles [9].

Fig. 4 shows the performance of initialization for the *rank* function by measuring the time taken by `rank_init` for inputs up to 1000Mbit. We make measurements for 1000 values of the input size. As expected, the result seems linear. There are several small gaps, for input size 537Mbit for example. This happens because the parameters for Jacobson's *rank* algorithm are changed at this point: sz_2 and `k` are changed from 30bit to 31bit, `w1` is changed from 29bit to 30bit. As a result, the size of the first-level directory decreases from 17.3Mbit to 16.8Mbit and the second-level directory, from 179Mbits to 174Mbits, leading to a shorter initialization time.

Benchmark Environment The operating system is Debian GNU/Linux 8.4 (Jessie) amd64 and the CPU is the Intel Core i7-4510U CPU (2.00GHz, Haswell). The time is measured using the `clock_gettime` function with the `CLOCK_PROCESS_CPUTIME_ID` resolution set to 1ns. The *rank* implementation is extracted by Coq 8.5pl1 and compiled to a native binary with `ocamlopt` version 4.02.3. C programs are compiled with gcc 4.9.2 with options `-O -march=native` (`-march=native` is used to enable POPCNT and LZCNT of recent Intel processors).

About OCaml's Garbage Collector `Gc.full_major` and `Gc.compact` are invoked before each measurement to mitigate the GC effect. Garbage collection does not occur during lookup measurements (`major_collections` and `minor_collections` in `Gc.stat` are unchanged). During initialization measurements, the GC has a small impact. Indeed, in Fig. 4, major garbage collection happens at most 226 times during one initialization measurement. Moreover, using another experiment with `gprof`, we checked that the time spent by the GC (with `Gc.full_major` and `Gc.compact` disabled) during the `rank_init` benchmark accounts for less than 5%.

7 Discussion and Perspectives

About Complexity For the time being, we limited ourselves to benchmarking the extracted code for time-complexity. It would be more convincing to perform formal verification using a monadic approach (e.g., [15]). We have addressed the issue of space-complexity in Sect. 5.3. In general, one may also wonder about the space-complexity of intermediate data structures. In this paper, we obviously did not build any but this could also be addressed by counting the number of cons cells using a monad.

About Extraction of Natural Numbers In this paper, there is no problem when we extract Coq `nat` to OCaml `int`, despite the fact that `nat` has no upper-bound. OCaml `int`s are $(w-1)$-bit signed integers that can represent positive integers less than 2^{w-2} (w is the number of bits in a machine word) [12]. However, the maximum number of bits in an OCaml `bytes` is $2^{w-10}w$ bits because one OCaml block may not contain more than 2^{w-10} words [12]. Since $2^{w-10}w < 2^{w-2}$ for $w = 32$ and $w = 64$, an `int` can always represent the number of bits in a `bytes`. For this reason, `nat` arguments of functions such as `bnth` or intermediate values in the *rank* algorithm do not overflow when turned into `int`. This can be ensured during formal verification by using a type for fixed-size integers (such as `int : nat -> Type` in [1]) instead of natural numbers.

About Alignment The extracted code can be further optimized by insisting on having the sizes (`w1`, `w2` in this paper) of the integers in the directories to be multiples of 8. This removes the need for masking an shifting when reading entries from the directories. This can be enforced by modifying `rank_default_param`.

About the Correctness of OCaml Code The OCaml part of the library has not been formally proved, but it has been extensively tested for functional correctness. We have implemented a test suite for the OCaml bitstring library using OUnit [17]. Concretely, we test functions for `bits` by comparison with list functions using random bitstrings. We also test the extracted *rank* function by comparison with the `rank` function defined in specification like style, i.e., `count_mem b (drop i s)`. Since we plan to reuse this library for other functions, it will endure even more testing. Formal verification of the OCaml part would be interesting, but it seems difficult as of today, because we are relying on unspecified features regarding optimization, `Obj.magic`, and C.

Our rank function is careful to use bitstrings in a linear way (i.e., it never adds bits twice to the same bitstring), but the correctness of the OCaml bitstring library does not rely on this fact. Whenever it detects repeated addition to a shared buffer, which can be seen through a discrepancy between the `used` field of the `bits_buffer` and the *len* part of the `Bref`, it copies the first *len* bits to a new buffer before adding the extra bits.

Formal verification of the Coq library may be used to further guarantee the time-complexity properties of the OCaml library. For example, to achieve linear-time construction of arrays with `bappend` (Sect. 4.2), `bappend s s'` must be called on `s` at most once. The approach that we are currently exploring to ensure this property is to augment the *rank* function with an appropriate monad.

About Performance of the Extracted Implementation We have not yet undertaken a thorough benchmark comparison with existing libraries for succinct data structures. This is mostly because our purpose in this paper is first and foremost verification, but also because the libraries we have checked so far do not seem to implement the same *rank* algorithm, making comparison difficult. Nevertheless, we can already observe that extracted OCaml code does not suffer from any significant performance loss compared to existing libraries. For example, we have observed that the SDSL [16] *rank* function for H_0-compressed vectors executes in about $0.1 \sim 1.8\mu s$ depending on algorithm's parameters while our *rank* function executed in $0.83\mu s$ (see Sect. 6.2). (To be fair, it is likely that our *rank* function consumes more memory since Jacobson's algorithm does not compress its input.) We believe that this is an indication that our approach can indeed deliver acceptable performance with the benefit of formal verification.

8 Conclusion

We discussed the verification of an OCaml implementation of the *rank* function for succinct data structures. We carried out formal verification in the Coq proof-assistant, from which the implementation was automatically extracted. We assessed not only functional correctness but also storage requirements, thus ensuring that data structures are indeed succinct. To obtain efficient code, we developed a new OCaml library for bitstrings whose interface match the Coq lists used in formal verification. To the best of our knowledge, this is the first application of formal verification to succinct data structures.

We believe that the libraries developed for the purpose of our experiment are reusable: the OCaml library for bitstrings of course, the array interface for directories (that are used by other functions for succinct data structures), lemmas developed for the purpose of formal specification of *rank* (as we saw in Sect. 5.2, verification of functional correctness can be carried out at the abstract level). We also discussed a number of issues regarding extraction from Coq to OCaml: the interplay between inlining at extraction-time and by the OCaml compiler, the soundness of code replacement at extraction-time, etc. Based on the results of this paper, we are now tackling formal verification of *rank*'s counterpart function *select* and plan to address more advanced algorithms.

Acknowledgments The authors are grateful to the anonymous reviewers for their helpful comments. This work is partially supported by a JSPS Grant-in-Aid for Scientific Research (Project Number: 15K12013).

References

1. R. Affeldt, N. Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In: ASIAN 2006. LNCS, vol. 4435, pp. 346–360. Springer, 2008.
2. R. Agarwal, A. Khandelwal, I. Stoica. Succinct: Enabling Queries on Compressed Data. In: NSDI 2015. pp. 337–350. USENIX Association, 2015. Technical report available at `http://people.eecs.berkeley.edu/~rachit/succinct-techreport.pdf`.
3. M. Armand, B. Grégoire, A. Spiwack, L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In: ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, 2010.
4. D. Clark. Compact Pat Trees. Doctoral Dissertation. University of Waterloo, 1996.
5. The Coq Development Team. Reference Manual. Version 8.5. Available at `http://coq.inria.fr`. INRIA (2004-2016).
6. Free Software Foundation. GCC 4.9.2 Manual. `http://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc`. 2014.
7. G. Gonthier, A. Mahboubi, E. Tassi. A Small Scale Reflection Extension for the Coq system. Version 16. Technical report RR-6455. INRIA, 2015.
8. Intel Advanced Vector Extensions Programming Reference. Jun. 2011.
9. Intel 64 and IA-32 Architectures Optimization Reference Manual. Sep. 2015.
10. Intel SSE4 Programming Reference. Apr. 2007.
11. G. Jacobson. Succinct static data structures. Doctoral Dissertation. Carnegie Mellon University, 1988.
12. R. W.M. Jones. A beginners guide to OCaml internals. `https://rwmj.wordpress.com/2009/08/04/ocaml-internals`. 2009.
13. D. K. Kim, J. C. Na, J. E. Kim, K. Park. Efficient Implementation of Rank and Select Functions for Succinct Representation. In: WEA 2005. LNCS, vol. 3503, pp. 315–327. Springer, 2005.
14. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood. seL4: formal verification of an operating-system kernel. Commun. ACM 53(6):107–115. 2010.
15. T. Nipkow. Amortized Complexity Verified. In: ITP 2015. LNCS, vol. 9236, pp. 310–324. Springer, 2015.
16. SDSL: Succinct Data Structure Library. `https://github.com/simongog/sdsl-lite`.
17. OUnit: Unit test framework for OCaml. `http://ounit.forge.ocamlcore.org/`.
18. D. Okanohara. The world of fast character string analysis. (In Japanese.) Iwanami Shoten, 2012.

19. A. Tanaka, R. Affeldt, J. Garrigue. Formal Verification of the rank Function for Succinct Data Structures. `https://staff.aist.go.jp/tanaka-akira/succinct/index.html`.

A OCaml Functions for Bitstrings

We equip the OCaml type `bits` (Sect. 4.2) with the same functions as the interface of the Coq type `bits` (Sect. 4.1), but so as to achieve the time-complexities required by Jacobson's *rank* function. Indeed, most OCaml functions that we propose as a replacement achieve the same tasks in constant-time instead of linear-time.

- `bsize` runs in constant-time because it just returns the first parameter of `Bref`.
- We implement `bnth` in constant-time easily by using OCaml functions for random-access to `bytes` (`Bytes.get`, `Bytes.set`).
- `bappend s s'` runs in $O(len')$-time for array construction. (len, len' are the lengths of `s`, `s'`.) More precisely, `bappend` works in $O(len')$-time if it is possible to append len' bits in the `bits_buffer` of `s` (i.e., when $len = used$ and $used + len' \leq 8 \times |\texttt{data}|$). In this case, `bappend` copies the content of `s'` into the `bits_buffer` of `s` by a destructive update and returns a newly allocated `bits` which length is $len + len'$. (If the buffer is not long enough, i.e., $used + len' > 8 \times |\texttt{data}|$, it is doubled but this doesn't change the time complexity with amortization.) If the destructive update is not possible, `bappend` copies the len bits of `s` into a newly allocated `bits_buffer`. This copy needs linear-time and space w.r.t. len. However, as far as the initialization phase of Jacobson's *rank* algorithm is concerned, the two arrays are constructed from left to right, so that `bappend` always runs in $O(len')$ with amortization.
- `bcount b i l s` runs in $O(l)$-time in general (the Coq `bcount` requires an additional $O(i)$ because of the `drop` function, whereas in OCaml access to the `i`th bit is direct). In fact, we have implemented `bcount` to use specialized assembly instructions when possible. Concretely, `bcount` is implemented in C to use gcc's `__builtin_popcountl` [6], which counts the number of bits set in a `long` value. For example, gcc generates POPCNT instructions for Intel SSE4.2 [10], so that we can assume that `__builtin_popcountl` works in constant-time.

B Core Part of the Extracted OCaml Code

```
let rank_lookup aux0 i =
  let b = aux0.query_bit in
  let param0 = aux0.parameter in
  let w1 = param0.w1_of in
  let w2 = param0.w2_of in
  let dirpair = aux0.directories in
  let j2 = (/) i (Pervasives.succ param0.sz2p_of) in
  let j3 = (mod) i (Pervasives.succ param0.sz2p_of) in
  let j1 = (/) j2 (Pervasives.succ param0.kp_of) in
  (+)
    ((+) (let s = fst dirpair in Pbits.getword (( * ) j1 w1) w1 s)
```

```
      (let s = snd dirpair in Pbits.getword (( * ) j2 w2) w2 s))
    (Pbits.bcount (Obj.magic b) (( * ) j2 (Pervasives.succ param0.sz2p_of))
      j3 aux0.input_bits)

let rank_init b s =
  let param0 = rank_param (Pbits.bsize s) in
  let w1 = param0.w1_of in
  let w2 = param0.w2_of in
  { query_bit = b; input_bits = s; parameter = param0; directories =
  (let rec buildDir j i n1 n2 d1 d2 =
     let m =
       Pbits.bcount (Obj.magic b)
         (( * ) ((-) param0.nn_of j) (Pervasives.succ param0.sz2p_of))
         (Pervasives.succ param0.sz2p_of) s
     in
     ((fun fO fS n -> if n=0 then fO () else fS (n-1))
        (fun _ ->
        let d1' = Pbits.bappend d1 (Pbits.bword w1 ((+) n1 n2)) in
        let d2' = Pbits.bappend d2 (Pbits.bword w2 0) in
        ((fun fO fS n -> if n=0 then fO () else fS (n-1))
           (fun _ -> (d1', d2'))
           (fun jp -> buildDir jp param0.kp_of ((+) n1 n2) m d1' d2')
           j))
        (fun ip ->
        let d2' = Pbits.bappend d2 (Pbits.bword w2 n2) in
        ((fun fO fS n -> if n=0 then fO () else fS (n-1))
           (fun _ -> (d1, d2'))
           (fun jp -> buildDir jp ip n1 ((+) n2 m) d1 d2')
           j))
        i)
   in buildDir param0.nn_of 0 0 0 Pbits.bnil Pbits.bnil) }
```

C Summary of the Implementation, Verification, Extraction, and Testing of the *rank* Algorithm

Figure 5 summarizes the experiment described in this paper. Relevant parts of implementation files are indicated for browsing (see the code online [19]).

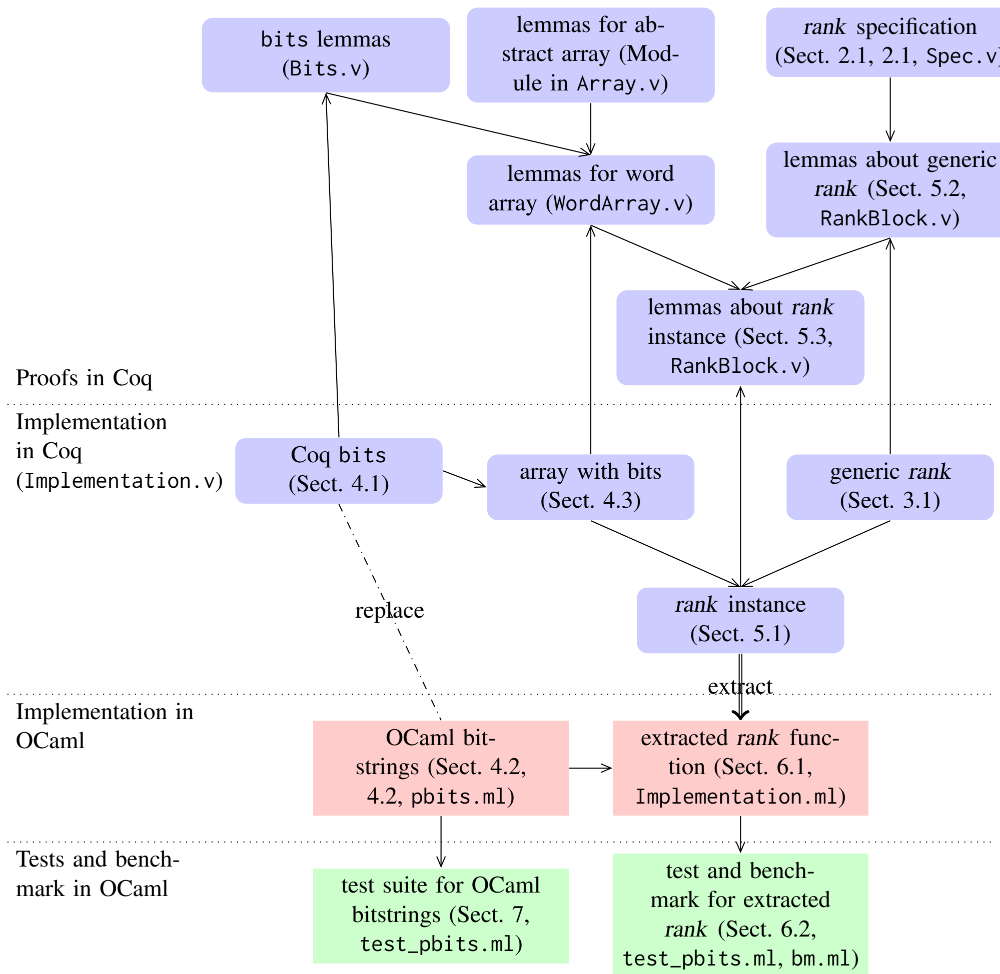

Fig. 5. Dependency graph for the verification of Jacobson's algorithm. Arrows $A \leftarrow B$ read as "A depends on B"

.